

How To Create AI Agents With Python From Scratch (Full Guide)

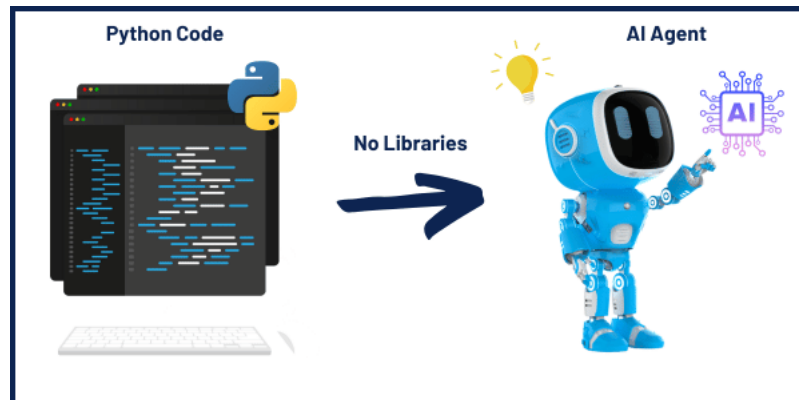


Hasan
April 30, 2024

2 Comments

In this post, we will create an Autonomous AI Agent With Python from Scratch.

We will **NOT** use any third-party libraries like Langchain or CrewAI; we will use **pure Python!**



We will start with a very basic example of an AI agent that helps you understand the basic structure and development process of AI agents.

From there, we will move to an **advanced real-world example**.

What is an AI Agent?

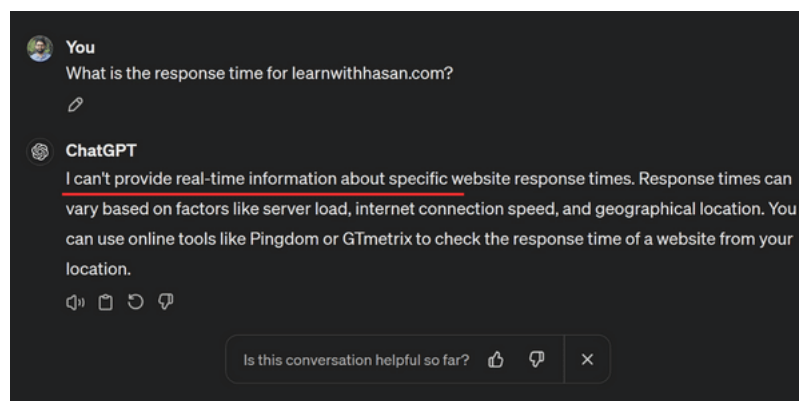
If we ask an AI like ChatGPT: **What is the response time for learnwithhasan.com?**

Do you think it can answer it?

If you said NO, you are right.

And if you said YES, you are also right!

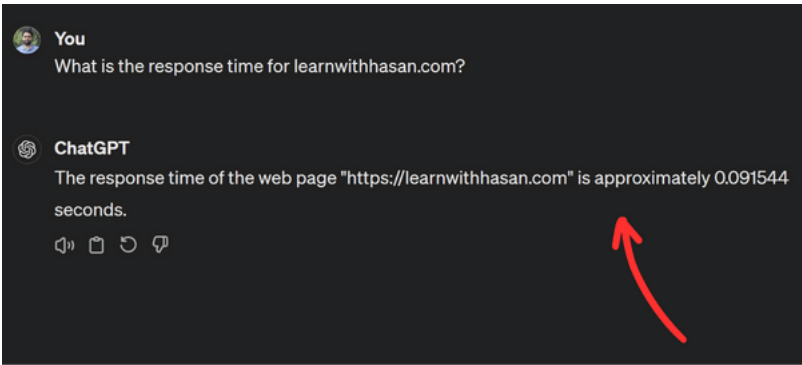
Interestingly, both answers could be considered correct. Here is ChatGPT's Answer to the question:



It was not able to answer!

As we learned in the [prompting engineering course](#), one of the main limitations of LLMs is that they can't access real-time data. They generate responses based solely on pre-existing training data.

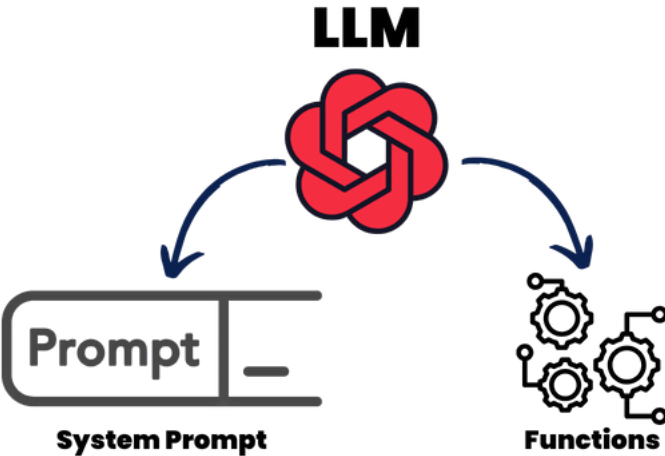
However, look at this now:



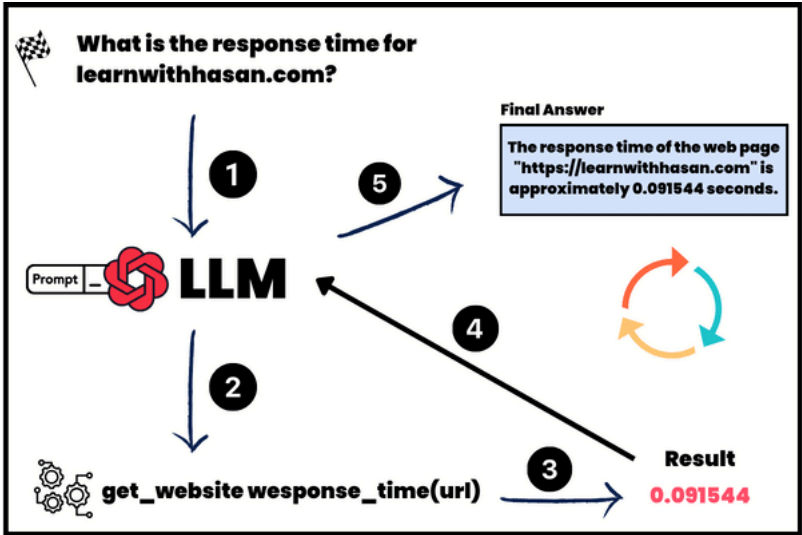
What happened ?!

Answer: Autonomous AI Agents 🦾

An Autonomous AI Agent integrates a large language model (LLM) with external functions and enhanced prompting mechanisms.



To understand the concept, let's see how the LLM was able to answer our question.



1- Query Input: First, we send our question to the LLM.

2- Processing with ReAct System Prompt: The LLM is Powered by a ReAct System Prompt that allows it to think about the question and how it should answer it. We call this a Thought. *We will discuss this more in the next sections.*

3- External Function Execution: The LLM then selects and executes an external function. In this case, `get_website_response_time(URL)`

4- Response Generation: After obtaining the real-time data, the AI crafts and delivers a response based on the result.

This integration of thinking, decision-making, and action mirrors human problem-solving processes, showcasing how AI can bypass traditional limitations.

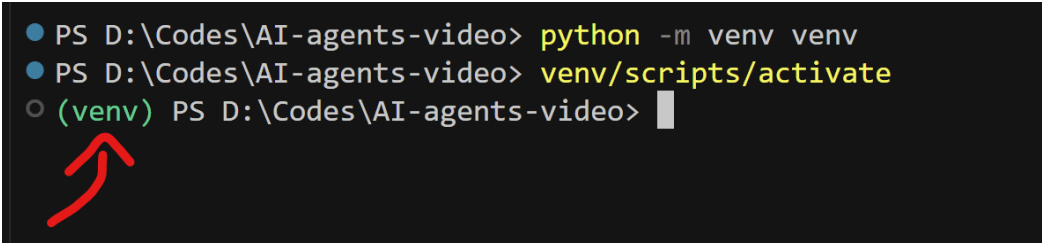
Getting Started

In this guide, we'll build AI agents from scratch using Python. Let's start by setting up a new Python project.

You can choose any IDE, but for this guide, I'll be using [Visual Studio Code](#).

Create and Activate a Virtual Environment

Open your terminal, create a new virtual environment, and activate it.



```
PS D:\Codes\AI-agents-video> python -m venv venv
PS D:\Codes\AI-agents-video> venv/scripts/activate
(venv) PS D:\Codes\AI-agents-video>
```

A red arrow points to the `(venv)` prompt in the terminal output.

Install the OpenAI Package

For this example, we'll use the OpenAI API as our large language model, although you could also use models from Anthropic, Gemini, or open-source models.

Ensure your API key is ready. Create a `.env` file in your project and add your key so

```
1. OPENAI_API_KEY = "sk-XX"
```

With the virtual environment active, install the OpenAI Python package:

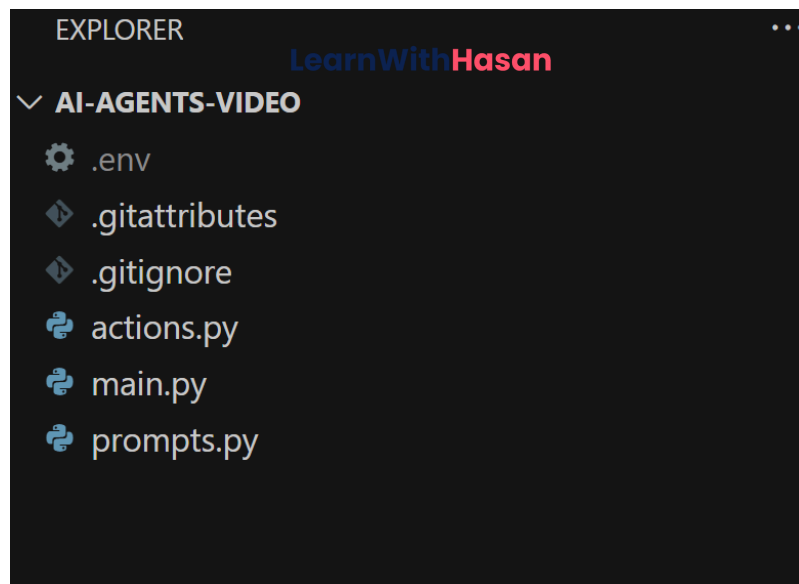
```
1. pip install openai
```

Done? Great 

Set Up Your Project Files

Create three Python files: `actions.py` , `prompts.py` , and `main.py` .

You should have now something like this:



Generate Text with OpenAI API

Open the `main.py` file and create a simple function to generate text using the OpenAI API. This function will power our AI agent:

Here is the code:

```
1. from openai import OpenAI
2. import os
3. from dotenv import load_dotenv
4.
5. # Load environment variables
6. load_dotenv()
7.
8. # Create an instance of the OpenAI class
9. openai_client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
10.
11.
12. def generate_text_with_conversation(messages, model = "gpt-3.5-turbo"):
13.     response = openai_client.chat.completions.create(
14.         model=model,
15.         messages=messages
16.     )
17.     return response.choices[0].message.content
```

This script loads your API key from the `.env` file and creates an instance of `OpenAI` to handle requests.

The `generate_text_with_conversation` function is straightforward, taking two parameters `model` and `messages` to generate a response.

Test Your Function

Before moving on, let's ensure everything is working as expected. Test the function by simulating a conversation:

```
1. # Define a list of messages to simulate a conversation
2. test_messages = [
3.     {"role": "user", "content": "Hello, how are you?"},
4.     {"role": "system", "content": "You are a helpful AI assistant"}
5. ]
6.
7. # Call the function with the test messages
8. response = generate_text_with_conversation(test_messages)
9. print("AI Response:", response)
```

Done? Perfect! ✅

Now that our basic setup is complete, we're ready to move on to the core parts of building our agent.

Define the Functions

In this part of the guide, we will specify the actions or functions that our AI agent can access. This enables the agent to utilize external functionalities when responding to user queries.

Create Basic Functionality

Open the `actions.py` file. Here, we'll define a simple function to simulate response times for different websites:

```
1. def get_response_time(url):
2.     if url == "learnwithhasan.com":
3.         return 0.5
4.     if url == "google.com":
5.         return 0.3
6.     if url == "openai.com":
7.         return 0.4
```

This dummy function returns fixed response times based on the provided URL. It serves as a basic example to help us understand how the agent can utilize external functions.

Understanding the Setup

By defining these functions, we establish a framework that the AI agent can refer to when needed.

This approach is crucial for integrating real-world functionalities into our agent, which we will explore more fully in later sections.

Next, we will go into another vital component of our AI agent: the **ReAct System Prompt**.

This will enhance the agent's ability to think and respond in a dynamic and context-aware manner.

The ReAct Prompt

The ReAct Prompt is what enables our AI agent to mimic human behavior.

This system prompt guides the model through a cycle of Thought, Action, and Response, allowing it to handle user queries effectively.

To put it simply, the ReAct prompt instructs the model to think about the user query, understand it, decide how to answer, pick an action if needed, and then use this to answer the question as best it can.

Let me share the prompt, and then I'll explain.

Define the ReAct Prompt

In the `prompts.py` file, add the following system prompt configuration:

```
1. system_prompt = """
2.
3.     You run in a loop of Thought, Action, PAUSE, Action_Response.
4.     At the end of the loop you output an Answer.
5.
6.     Use Thought to understand the question you have been asked.
7.     Use Action to run one of the actions available to you - then return PAUSE.
8.     Action_Response will be the result of running those actions.
9.
10.    Your available actions are:
11.
12.    get_response_time:
13.    e.g. get_response_time: learnwithhasan.com
14.    Returns the response time of a website
15.
16.    Example session:
17.
18.    Question: what is the response time for learnwithhasan.com?
19.    Thought: I should check the response time for the web page first.
```

```

20.     Action:
21.
22.     {
23.         "function_name": "get_response_time",
24.         "function_params": {
25.             "url": "learnwithhasan.com"
26.         }
27.     }
28.
29.     PAUSE
30.
31.     You will be called again with this:
32.
33.     Action_Response: 0.5
34.
35.     You then output:
36.
37.     Answer: The response time for learnwithhasan.com is 0.5 seconds.
38.     """

```

This system prompt instructs the LLM to run within a loop of Thought, Action, and Action_Response.

The loop structure (Thought, Action, PAUSE, Action_Response) guides the LLM:

- **Thought:** Understand and interpret the query.
- **Action:** Select and execute the appropriate function from the available actions.
- **Action_Response:** Use the result from the action to formulate the response.

Available actions

Then, we tell the LLM what actions are available, showing a simple example with the parameter and a simple description so the model understands the function.

```

Your available actions are:

get_response_time:
e.g. get_response_time: learnwithhasan.com
Returns the response time of a website

```

💡 **Make sure to match the function name with the one you defined in Python.**

Example session

Then, we show the LLM an example of how it will act to answer a sample query.

The most important part here is how it will **return the Action**:

```

Action:

{
    "function_name": "get_response_time",
    "function_params": {
        "url": "learnwithhasan.com"
    }
}

```

You can see here that I instructed the LLM to return the action in a **JSON Format**.

This will help us later work with functions and run them, as you will in the last part when we put things together.

Why a Loop?

This looping mechanism mimics the steps an LLM takes: understanding the question, taking an action based on that understanding, and using the outcome of the action to respond.

This process can range from a couple of loops for simple tasks to potentially hundreds for more complex scenarios.

Putting Things Together

Having established the ReAct System Prompt and defined the necessary functions, we can now integrate these elements to construct our AI agent.

Let's return to our `main.py` script to complete the setup.

Define Available Functions

First, list the functions the agent can utilize. For this example, we only have one:

```
1. available_actions = {
2.     "get_response_time": get_response_time
3. }
```

In our case, we have only one function.

This will enable the agent to select the correct function efficiently.

Set Up User and System Prompts

Define the user prompt and the messages that will be passed to the `generate_text_with_conversation`, the function we previously created:

```
1. user_prompt = "What is the response time for learnwithhasan.com?"
2.
3. messages = [
4.     {"role": "system", "content": system_prompt},
5.     {"role": "user", "content": user_prompt},
6. ]
```

The system prompt, structured as a ReAct loop directive, is provided as a system message to the OpenAI LLM.

Now, OpenAI's LLM Model will be instructed to act in a loop of Thought. Action and Action Result!

Create the Agentic Loop

Implement the loop that processes user inputs and handles AI responses:

```
1. turn_count = 1
2. max_turns = 5
3.
4.
5. while turn_count < max_turns:
6.     print(f"Loop: {turn_count}")
7.     print("-----")
8.     turn_count += 1
9.
10.    response = generate_text_with_conversation(messages, model="gpt-4")
11.
12.    print(response)
13.
14.    json_function = extract_json(response)
15.
16.    if json_function:
17.        function_name = json_function[0]['function_name']
18.        function_parms = json_function[0]['function_parms']
19.        if function_name not in available_actions:
20.            raise Exception(f"Unknown action: {function_name}: {function_parms}")
21.        print(f"-- running {function_name} {function_parms}")
22.        action_function = available_actions[function_name]
23.        #call the function
24.        result = action_function(**function_parms)
25.        function_result_message = f"Action_Response: {result}"
26.        messages.append({"role": "user", "content": function_result_message})
27.        print(function_result_message)
28.
29.    else:
30.        break
```

This loop reflects the ReAct cycle, generating responses, extracting JSON-formatted function calls, and executing the appropriate actions.

So, we generate the response, and we check if the LLM returned a function.

I created the `extract_json` method to make it easy for you to extract any functions from the LLM response. **LearnWithHasan**

In the following line:

```
1. json_function = extract_json(response)
```

We will check if the LLM returned a function to execute; if yes, it will execute and append the result to the messages so that in the next turn, the LLM can use the `Action_Response` to answer the user's query.

Test the Agent

To see this agent in action, you can download the complete codebase using the link provided below:

Basic AI Agent Code

SEO Auditor AI Agent

Now, after you have learned how to build AI agents from scratch using our primary example.

Let's advance to a more practical example by creating an SEO Auditor AI Agent. This agent will demonstrate real-world utility and showcase the adaptability of our initial setup.

Define a New Function

To begin, we'll define a function for SEO auditing in the `actions.py` file:

```
1. from SimplerLLM.tools.rapid_api import RapidAPIClient
2.
3. def get_seo_page_report(url :str):
4.     api_url = "https://website-seo-analyzer.p.rapidapi.com/seo/seo-audit-basic"
5.     api_params = {
6.         'url': url,
7.     }
8.     api_client = RapidAPIClient()
9.     response = api_client.call_api(api_url, method='GET', params=api_params)
10.    return response
```

💡 If you are interested in seeing an AI agent example with multiple function, you can check [my video library here](#).

Anyway, This function utilizes an **SEO Audit API** to generate a detailed report for any specified website or webpage, providing valuable insights into SEO performance.

So, the AI Agent can use this report to answer any question related to the target web page.

I used **SimplerLLM** Library here, which has the `RapidAPIClient` built-in and allows us to call any API on RapidAPI easily. **Imagine how many functionalities you can give to your AI Agent with this one function!**

Update the System Prompt

```
1. react_system_prompt = """
2.
3. You run in a loop of Thought, Action, PAUSE, Action_Response.
4. At the end of the loop you output an Answer.
5.
6. Use Thought to understand the question you have been asked.
7. Use Action to run one of the actions available to you - then return PAUSE.
8. Action_Response will be the result of running those actions.
9.
10. Your available actions are:
```




```
11.
12. get_seo_page_report:
13. e.g. get_seo_page_report: learnwithhasan.com
14. Returns a full seo report for the web page
15.
16.
17. Example session:
18.
19. Question: is the heading optimized for the keyword "marketing" in this web page: learnwithhasan.com?
20. Thought: I should generate a full seo report for the web page first.
21. Action:
22.
23. {
24.   "function_name": "get_seo_page_report",
25.   "function_params": {
26.     "url": "learnwithhasan.com"
27.   }
28. }
29.
30. PAUSE
31.
32. You will be called again with this:
33.
34. Action_Response: the full SEO report
35.
36. You then output:
37.
38. Answer: Yes, the heading is optimized for the keyword "marketing" in this web page since the SEO
    report shows that the keyword is in the H1 heading.
39.
40. """strip()
```

You can see here that I changed the available functions and the example session to match our AI agent Role.

Adjust Main Agent File

Finally, update the `main.py` to incorporate the new action:

```
1. available_actions = {
2.     "get_seo_page_report": get_seo_page_report
3. }
```

Run the SEO Audit Agent

You see now how easy it is to build your AI Agents!

Define the function, update the prompt, and run!

Example of running the SEO Auditor AI Agent:

If we ask now about the response time for learnwithhasan.com

We will get an answer:

```
The response time of the web page "https://learnwithhasan.com" is approximately 0.091544
seconds.
```

Now, it is a real number, as it was obtained from the SEO report.

You can even ask it a more generic question like:

“Suggest some SEO optimization tips for learnwithhasan.com”

And we will get an answer:

- LearnWithHasan
1. Add alt tags to images: There are 4 images on the website that are missing alt tags. Adding descriptive alt tags to these images can improve accessibility and provide a ranking boost.
 2. Improve internal linking: The website has 49 total links, but only 13 of them are internal. Adding more internal links can help spread link equity around your website and can boost the SEO of more pages.
 3. Optimize page headings: There is only one H1 heading. You can consider using more H2, H3, and H4 headings for better content hierarchy and easier readability.
 4. Increase content length: The website has a total word count of 707. Longer, in-depth content tends to rank better in search engines.
 5. Implement hreflangs: The website doesn't use hreflangs. If the site has content in multiple languages, hreflangs can help search engines understand the language and geographical targeting of a webpage.

These responses, based on actual SEO reports, showcase the agent’s ability to provide expert SEO advice.

I hope you enjoyed this guide and learned something new today!

For a more in-depth exploration and additional examples, consider my course, **[“Build AI Agents From Scratch With Python.”](#)**

Remember, If you have any questions or encounter issues, I’m available nearly every day on the [forum](#) to assist you—for free!

And if you like to see all this in action, you can check this free video:

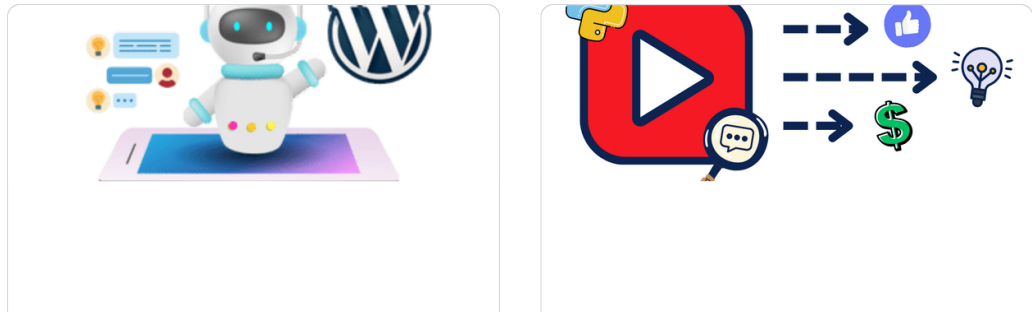
Create AI Agents From Scratch With Python! (Free Course)



Categories: [AI](#), [ChatGPT](#), [OpenAI](#), [Prompt Engineering](#), [Python](#), [Research](#)



Related Articles



How To Create a Free AI Chatbot on WordPress From Scratch!

LearnWithHasan

In this post, I will show you how to create a Free AI Chatbot on a WordPress site WITHOUT using third-party services or chatbot plugins....



Hasan
April 23, 2024

💬 15

Extract & Analyze YouTube Comments With Python and YouTube API To Boost Your Marketing!

In this post, I will show you how to extract YouTube comments from your channel and turn them into valuable data for your marketing campaigns....



Husein
August 30, 2024

💬 0

Responses

Your email address will not be published. Required fields are marked *

Write a response...

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.



I'm not a robot

reCAPTCHA
[Privacy](#) - [Terms](#)

Publish



Dharani R
May 3, 2024

hello, i m looking for inculcating an agent to my lung cancer detection model using cnn.
please help me able to achieve it

Reply.



Hasan
May 8, 2024

please join us on the forum, so we can follow up properly

Reply.

